Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Rok Novosel

Generiranje algoritmov za štetje k-gramov

diplomsko delo

univerzitetni študijski program
prve stopnje
računalništvo in informatika

Mentor: doc. dr. Jurij Mihelič

Ljubljana, 2016

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:
Generiranje algoritmov za štetje k-gramov

Tematika naloge:

Inženiring algoritmov se ukvarja z razvojem algoritmov, ki so uporabni in učinkoviti v praksi, zato je eden izmed glavnih poudarkov področja eksperimentalno preizkušanje algoritmov. Namen diplomske naloge je uporabiti tehnike generiranja algoritmov in jih preko pristopov s področja inženiringa algoritmov preizkusiti in ovrednotiti. V okviru diplomske naloge na kratko preglejte področje, nato izberite primeren problem, za katerega bi bilo smiselno generirati algoritme. Implementirajte ustrezne generatorje in eksperimentalno ovrednotite njihovo uporabnost v praksi. Generirane algoritme primerjajte s splošnim algoritmom za izbrani problem. Implementacijo izvedite v obliki splošnonamenske algoritmične knjižnice.

Za strokovno podporo in napotke se iskreno zahvaljujem mentorju doc. dr. Juriju Miheliču. Za moralno podporo gre velika zahvala moji punci Tjaši in moji celotni družini.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
ASCII	American Standard Code for Information Interchange	Ameriška standardna koda za izmenjavo informacij
RAM	Random Access Machine	Stroj z naključnim dostopom
PRAM	Parallel Random Access Machine	Paralelni stroj z naključnim dostpom
CPU	Central processing unit	Centralna procesna enota

Povzetek

Naslov: Generiranje algoritmov za štetje k-gramov

Avtor: Rok Novosel

Generativna algoritmika je proces zasnove in implementacije generiranih algoritmov. Spada v večje področje inženiringa algoritmov, ki se ukvarja z eksperimentalno evaluacijo in pohitritvijo algoritmov. Na začetku diplomske naloge opišemo oba področja in prikažemo osnove generativne algoritmike na primeru računanja predpon. Za glavni problem, ki ga bomo rešili z generativno algoritmiko smo si izbrali štetje k-gramov. Na začetku implementiramo splošni algoritem za štetje k-gramov, s katerim bomo primerjali generirane algoritme. Sledila je zasnova in implementacija osnovnega in izboljšanega generiranega algoritma. V zadnjem delu smo opravili eksperimentalno evaluacijo vseh algoritmov in preverili do katere meje so učinkoviti generirani algoritmi. V zaključku je diskusija rezultatov, možne izboljšave in možnosti za nadaljno raziskovanje.

Ključne besede: algoritmi, generiranje algoritmov, k-grami, računanje predpon, končni avtomati stanj.

Abstract

Title: Generating algorithms for counting k-grams

Author: Rok Novosel

Generative algorithmics is a process of designing and implementing generated algorithms. It belongs in a larger area of algorithm engineering, which deals with experimental evaluation and speedup of algorithms. At the beginning of the thesis we describe both areas and show the basics of generative algorithmics solving parallel prefix scan. For our main problem we choose counting k-grams. First, we implement a general algorithm for counting k-grams against which we will compare the generated algorithms. This was followed by the design and implementation of basic and enhanced generated algorithm. At the end we conducted an experimental evaluation of all the algorithms and verify the effectiveness of generated algorithms. In the last section we discussed the results, potential improvements and opportunities for further exploration.

Keywords: algorithm, generating algorithms, k-grams, prefix scan, finite state automatons.

Poglavje 1

Uvod

Algoritmi in podatkovne strukture so osrednji del vsake računalniške aplikacije in zato tudi posredno pomemben del našega vsakdana. Hitro iskanje po velikih zbirkah podatkov, analiziranje človeškega genoma in težko zlomljiva kriptografija ne bi bili mogoči, če ne bi uporabljali sofisticiranih algoritmov. Zato je njihov razvoj ključnega pomena - ne samo za računalništvo, ampak za vsa področja, ki jih uporabljajo. Področje teoretičnega računalništva je v zadnjem času pokazalo večje zanimanje za reševanje problemov, ki nastajajo v realnih aplikacijah, vendar raziskovalci že desetletja razvijajo algoritme na klasičen način z uporabo matematičnih metod za ocenjevanje in napovedi obnašanja. Zahteva po algoritmih za uporabo v praktične namene zahteva prenovo teoretičnega pristopa tako, da bo podprt z eksperimentalnimi ugotovitvami. Potrebno je tudi uskladiti splošne modele, kot je RAM in obstoječe računalniške arhitekture, ker RAM ne predvideva pomembnih dejavnikov, kot je hierarhija pomnilnika, kompleksnost komunikacije, numerične natančnosti in uporabe hevristik. Prav zato se asimptotično hitri algoritmi lahko slabo obnesejo na realnih problemih, ker skrivajo velike konstante.

Cilj diplomske naloge je demonstracija uporabnosti generativne algoritmike na problemu štetja k-gramov. V razdelku 2 predstavimo področji inženiringa algoritmov in generativne algoritmike. V podrazdelku 2.3 prikažemo osnove generativne algoritmike na primeru računanja predpon. V razdelku 3

bo predstavljen problem štetja k-gramov, naša zasnova osnovnega in izboljšanega algoritma ter implementacija obojega. Prikazana je tudi uporaba generirane knjižnice. Rezultate analize generiranja in izvajanja algoritmov iz razdelka 3 najdemo v razdelku 4. V zadnjem razdelku 5 analiziramo rezultate in potencialno uporabo generativne algoritmike za ostale probleme.

Poglavje 2

Generativna algoritmika

2.1 Inženiring algoritmov

Obstaja veliko modelov za razvijanje programske opreme, ki natančno opisujejo njen razvoj in analizo. Eden izmed najbolj znanih je slapovni model, ki opisuje zaporedni proces razvoja, implementacije, verifikacije in vzdrževanja programske opreme. Inženiring algoritmov namesto zaporedja korakov predlaga cikel, kar omogoča večkratno iteracijo in tako inkrementalno izboljšavo algoritma. Potek inženiringa algoritmov je sestavljen iz naslednjih korakov: [4]:

- specifična aplikacija,
- realistični model,
- zasnova,
- analiza,
- performančne zagotovitve,
- implementacija,
- poskusi,

- dejanski podatki,
- knjižnica algoritmov.

Začnemo s specifično aplikacijo, za katero poskušamo najti realistični model tako, da bodo rešitve, ki jih bomo našli, čim bolje ustrezale začetnim zahtevam. Glavni cikel procesa se začne z zasnovo algoritma, ki nakaže rešitev problema. Glede na to zasnovo analiziramo algoritem iz teoretičnega vidika (asimptotična analiza), da lahko preko tega dobimo določena performančna zagotovila. Naslednji korak je dejanska implementacija algoritma. Velikokrat je dolgotrajen in podcenjen, ampak igra ključno vlogo v tem procesu. Namreč, pogosto so razviti čisto teoretični algoritmi, kjer ni popolnoma jasno, kako jih pretvoriti v dejansko programsko kodo.

Algoritmi velikokrat na vhodu predpostavijo statično, vnaprej procesirano podatkovno strukturo, kjer se poizvedbe izvajajo v asimptotično hitrih časih. Vendar se v praktičnih scenarijih zgodi, da moramo podatke hraniti v bolj fleksibilnem formatu, npr. povezan seznamu namesto tabele. Ko imamo opravka z grafi, je to še toliko bolj očitno, ker jih shranjujemo kot dinamične grafe s seznami sosedov ali kot stisnjeno matriko sosedov. Take podatkovne strukture pa niso primerne za učinkovite poizvedbe. Za poizvedbe na grafih izvedene v konstantem času, bi potrebovali polno matriko sosedov. Ampak matrika, ki bi bila potrebna, da algoritem doseže optimalni čas, je enostavno prevelika, da bi jo lahko hranili v delovnem pomnilniku. To lahko privede do tega, da je asimptotično počasnejši algoritem boljši od asimptotično hitrejšega.[12].

Take vpoglede v algoritem lahko dobimo z eksperimentiranjem, ki je zadnji korak v našem glavnem ciklu. Ne zanimajo nas eksperimenti na naključni množici podatkov, tako kot bi testirali tradicionalno, ampak hočemo testirati na realnih podatkih, za katere je bil algoritem tudi zasnovan. Ta korak nam omogoča, da testiramo več različnih zasnovanih algoritmov na podatkih, ki so relevantni za praktično uporabo. Tako lahko bolje ocenimo konstante, ki so bile skrite v asimptotični analizi. S tem ocenimo tudi najboljši algoritem ne glede na to, če so vsi imeli iste asimptotične ocene.

Cilj je uporabiti znanje, pridobljeno v korakih analize, implementacije in eksperimentacije, da najdemo boljše zasnove algoritmov. Če naši eksperimenti pokažejo linearni izvajalni čas za algoritem, ki ima asimptotično kvadratni izvajalni čas, je to znak, da lahko izboljšamo našo teoretično analizo povprečnega izvajalnega časa [10]. Odkrijemo lahko tudi ozka grla s profiliranjem programske kode. Praktično uporabnost algoritma izboljšamo tako, da popravimo njegovo implementacijo ali pa zasnovo samega algoritma. Končni produkt cikla je običajno algoritemska knjižnica, ki je pripravljena za uporabo in nadaljnjo raziskavo. Prosto dostopna knjižnica omogoča lažje preverjanje znanstvenih odkritij in primerjavo med različnimi implementacijami. Taka knjižnica tudi pomaga zbližati akademsko teorijo in uporabnike v neakademskem svetu. Ti jo bodo pa bolj verjetno uporabili, če je opremljena z dobro strukturirano dokumentacijo.

2.2 Generativna algoritmika

Generativna algoritmika je področje znotraj inženiringa algoritmov in se ukvarja z generiranjem algoritmov in algoritmičnih knjižnic. Razlog za generiranje algoritmov je v tem, da večini primerov ne potrebujemo splošnega algoritma, ampak zelo specifične algoritme, ki delajo na majhni podmnožici problema. S tem želimo izboljšati izvajalni čas in zmogljivost algoritmov.

Osnovni pristop pri generativni algoritmiki je generiranje poizvedbenih podatkovnih struktur. Te strukture algoritmi naložijo v delovni pomnilnik in uporabljajo že izračunane podatke med samim izvajanjem namesto, da jih algoritem izračunava sproti. S tem zmanjšamo izvajalni čas algoritma, vendar povečamo količino delovnega pomnilnika, ki ga potrebujemo. V vsakem primeru je treba najti kompromis med pomnilnikom in izvajalnim časom s pomočjo eksperimentov na široki množici vhodnih podatkov. S tem lahko najdemo točko preloma, v kateri se splača uporabljati prebrano poizvedbeno podatkovno strukturo ali izračunavati vrednosti med izvajanjem algoritma. Vmesno rešitev, ki med izvajanjem algoritma hrani izračunane rezultate in

jih shrani za kasnejše poizvedovanje, imenujemo dinamično programiranje. [5, Poglavje 15].

Generativna algoritmika se še posebej prilega deli-in-vladaj algoritmom in algoritmom, ki so osnovani na končnih avtomatih stanj. Pri deli-in-vladaj algoritmih, ki so rekurzivne narave, lahko zaustavitveni pogoj nastavimo prej in zadnjih nekaj nivojev rekurzije sploščimo v en sam nivo. Pri algoritmih osnovanih na končnih avtomatih stanj pa lahko generiramo prehode stanj avtomata in morebitne ostale informacije v primerno podatkovno strukturo, npr. `class` v programskem jeziku Java ali `struct` v programskem jeziku C. Sam algoritem pa zastavimo tako, da sprejme generirani končni avtomat stanj, ki ga izvede za dane ostale parametre.

Še ena optimizacijska tehnika, ki jo že uporabljajo prevajalniki, je razvijanje zank [2]. Na podoben način bi lahko generirali vse iteracije zank. Ta tehnika podaljšuje kodo in s tem tudi čas potreben za prevajanje algoritma, ampak prihranimo na času, ki je potreben za skok na začetek zanke po koncu vsake iteracije.

2.3 Računanje predpon

Kot prvi problem za demonstracijo generativne algoritmike smo si izbrali problem paralelnega računanja predpon. Rešitev problema bomo predstavili teoretično in izpustili implementacijo. Računanje predpon je uporabno za leksikalno primerjanje nizov, radix sort, regularne izraze in še mnogo ostalih problemov [3].

2.3.1 Definicija problema

Definicija 1. Operacija računanja predpon kot argumenta vzame poljubno asociativno binarno operacijo in zaporedje števil

$$x = [x_0, x_1, x_2, \ldots, x_{n-1}]$$

dolžine n in vrne seznam izračunanih predpon

$$y = [x_0, (x_0 \oplus x_1), (x_0 \oplus x_1 \oplus x_2), \ldots, (x_0 \oplus x_1 \oplus \ldots \oplus x_{n-1})].$$

Oglejmo si primer. Za seznam števil $[5, 3, 4, 12, 1]$ in binarno operacijo seštevanja $\oplus = +$ dobimo naslednje izračunane predpone: $[5, 8, 12, 24, 29]$.

Implementacija zaporednega algoritma je trivialna in je prikazana v algoritmu 1. Preprosto iteriramo čez vse elemente vhodnega zaporedja x in z binarnim operatorjem združimo vrednost trenutnega elementa iz vhodnega zaporedja in prejšnjega elementa iz izhodnega zaporedja y. V splošnem bo algoritem naredil $n - 1$ korakov, kar pomeni, da ima časovno zahtevnost $O(n)$.

Algoritem 1 Zaporedno računanje predpon

```
1: y[0] = x[0]
2: for i in (1 to n) do
3:     y[i] ← y[i-1] ⊕ x[i]
4: end for
```

2.3.2 Zasnova vzporednega algoritma

Da bi lahko paralelno računali predpone je potrebno zaporedni algoritem bistveno spremeniti. Razlog leži v tem, da so iteracije glavne zanke algoritma medsebojne odvisne in ni lahkega načina za paralelizacijo. Sledi zasnova paralelnega algoritma za računanje predpon tabel velikosti, ki so potence števila 2.

Algoritem smo zasnovali kot ackiličnI usmerjeni graf $G = (V, E)$, kjer je V množica vozlišč, ki vsebuje $v_{i,j}$ za $i \in \{0, 1, \ldots, \log_2(n) - 1\}$ in $j \in \{0, 1, \ldots, \frac{n}{2} - 1\}$. To skupno znese $\frac{n \log_2 n}{2}$ vozlišč. Za vhodne povezave vozlišč velja $deg^-(v) = 2$ in za izhodne povezave velja $deg^+(v) = 1$. Vozlišče preko vhodnih povezav dobi izračunana rezultata, ki ju nato združi s podano binarno operacijo in jo poda naslednjemu vozlišču preko edine izhodne

povezave. Na začetku algoritma inicializiramo vozlišča $v_{0,j}$ na naslednji način

$$v_{0,j} = x_{2j} \oplus x_{2j+1}$$

za vsak j. Za vozlišča kjer $i \geq 1$ smo morali izpeljati posebni formuli. Za ta vozlišča smo izpeljali formuli za vrednosti, ki pridejo po levi vhodni povezavi in po desni vhodni povezavi. Formula za izračun levega vozlišča se nahaja v enačbi 2.1. Formula v vsakem primeru vrne vozlišče $v_{k,l} \in V$.

$$left(i,j) = v_{k,l}; k = i-1, l = 2^{i-1} + 2^i \left\lfloor \frac{j}{2^i} \right\rfloor - 1 \tag{2.1}$$

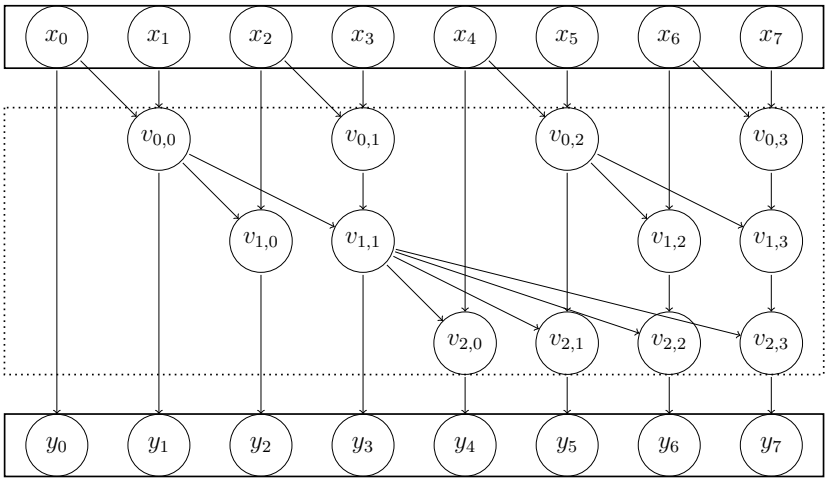
Formula za izračun desnega vozlišča se nahaja v enačbi 2.2. V primeru, da $j = 2^i \left\lfloor \frac{j}{2^i} \right\rfloor$ formula vrne element iz vhodnega zaporedja x. Za vse ostale vrednosti j vrne vozlišče $v_{k,l} \in V$.

$$right(i,j) = \begin{cases} x_m; m = 2j + 2^i, & \text{če } j = 2^i \left\lfloor \frac{j}{2^i} \right\rfloor \\ v_{k,l}; & \\ k = \left\lfloor \log_2(j - 2^i \left\lfloor \frac{j}{2^i} \right\rfloor) \right\rfloor, & \\ l = 2^{i-1} - 2^k + j, & \text{sicer} \end{cases} \tag{2.2}$$

Ko izvedemo celotni algoritem in izračunamo vrednosti vseh vozlišč, dobimo izračunane predpone s pomočjo formule v enačbi 2.3.

$$y[i] = \begin{cases} x_0, & \text{če } i = 0 \\ v_{k,l}; k = \lfloor \log_2(i) \rfloor, l = i - 2^k, & \text{sicer} \end{cases} \tag{2.3}$$

Na sliki 2.1 je prikazan graf, kjer $n = 8$.



Slika 2.1: Graf za izračun predpon zaporedja z osmimi elementi

2.3.3 Generiranje algoritma

Sledi prikaz generiranja algoritma za izračun predpon. Odločili smo se, da bo naš generirani algoritem živel v funkciji imenovani calculate_prefix_scan. Ker je ne moremo implementirati direktno, lahko zanjo pripravimo predlogo, ki jo bo uporabil generator pri generiranju. Predloga je prikazana v izvorni kodi 2.1.

```
function calculate_prefix_scan(x, op) {
  y = int[{{n}}]
  v = int[{{depth}}][{{length}}]
  {{assignments}}
  return y
}
```

Izvorna koda 2.1: Predloga za funkcijo calculate_prefix_scan

Funkcija kot argumenta sprejme vhodno tabelo x in funkcijo op, ki sprejme dva argumenta in vrne rezultat. Na mestu kjer je zapisano {{var}} bo generator zapisal spremenljivko z imenom var. Funkcija na začetku inicializira tabelo, kjer se bodo hranili rezultati izračunanih predpon. Nato inicializiramo dvodimenzionalno tabelo v, ki bo predstavljala vozlišča grafa iz prejšnjega

razdelka. Sledi zaporedje ukazov, ki bodo napolnili tabelo v in iz nje dobili izračunane predpone shranjene v tabeli y.

V algoritmu 2 implementiramo formuli iz enačb 2.1 in 2.2. Obe vrneta niz, ki predstavlja element iz tabele v ali element iz vhodne tabele x.

Algoritem 2 Pomožni funkciji za izračun vozlišč

```
1: function LEFT(i, j)
2:     k ← i-1
3:     l ← 2^(i-1) + 2^i ⌊ j / 2^i ⌋ − 1
4:     return "v[$k][$l]"
5: end function
6: function RIGHT(i, j)
7:     if j == 2^i ⌊ j / 2^i ⌋ then
8:         m ← 2j + 2^i
9:         return "x[$m]"
10:    else
11:        k ← ⌊ log_2(j − 2^i ⌊ j / 2^i ⌋) ⌋
12:        l ← 2^(i-1) − 2^k + j
13:        return "v[$k][$l]"
14:    end if
15: end function
```

Algoritem 3 kot argumenta sprejme dimenziji tabele v. Kot prikazano v prejšnjem razdelku, na začetku generiramo ukaze, ki bodo inicializirali prvo vrstico tabele v. Nato s pomočjo pomožnih funkcij left in right izračunamo ukaze, ki bodo generirali ukaze za ostale vrstice. Zaključimo z ukazi, ki bodo napolnili tabelo z izračunanimi predponami y.

V algoritmu 4 pa povežemo celotno rešitev. Algoritem kot argument sprejme velikost vhodne tabele n in iz tega izračuna število vrstic in stolpcev tabele v. Obe dimenziji uporabi, da s pomočjo funkcije generatePrefixScanAssignments izračuna ukaze za izračun predpon. S klicem funkcije writeTemplate vzamemo predlogo iz izvorne kode 2.1 in vanjo vstavimo velikost vhodne tabele n, število vrstic depth in število stolpcev length tabele v ter niz ukazov assignments, ki bodo izračunali predpone. Končni rezultat je nova datoteka, ki vsebuje funkcijo pripravljeno za prevajanje in uporabo.

Večje kot bi bilo število elementov, večja bi postajala tabela v in potrebovali bi veliko več ukazov, da bi jo napolnili. Tukaj bi morali skleniti

Algoritem 3 Algoritem, ki izračuna in vrne ukaze za izračun predpon

```
1: function generatePrefixScanAssignments(depth, length, n)
2:     assignments ← []
3:     for j in (0 to length) do
4:         left_vertex ← 2j
5:         right_vertex ← 2j+1
6:         assignments += "v[0][$j]=op(x[$left_vertex],x[$right_vertex])"
7:     end for
8:     for i in (1 to depth) do
9:         for j in (0 to length) do
10:            left_vertex ← left(i, j)
11:            right_vertex ← right(i, j)
12:            assignments += "v[$i][$j]=op($left_vertex, $right_vertex)"
13:        end for
14:    end for
15:    assignments += "y[0]=x[0]"
16:    for i in (0 to n) do
17:        k ← ⌊log_2 i⌋
18:        l ← i − 2^k
19:        assignments += "y[$i]=v[$k][$l]"
20:    end for
21:    return assignments
22: end function
```

Algoritem 4 Končni algoritem, ki generira algoritem za računanje predpon

```
1: function generatePrefixScanAlgorithm(n)
2:     depth ← log_2 n
3:     length ← n/2
4:     assignments ← generatePrefixScanAssignments(depth, length, n)
5:     writeTemplate(n, depth, length, assignments)
6: end function
```

kompromis med prostorom za shranjevanje algoritma in čas potreben za izvajanje algoritma. Najlažje bi ga našli z eksperimenti za velik nabor velikosti vhodnih seznamov. V neki točki se nam bi bolj splačalo, da algoritem sproti izračunava vrednosti levega in desnega vozlišča. Algoritem ne bi več deloval tako hitro, prihranili bi pa na prostoru za hranjenje algoritma, času prevajanja in na prostoru porabljenem v delovnem pomnilniku. Graf je bil zasnovan tako, da so vsi stolpci v določeni vrstici med sabo neodvisni in jih lahko izračunamo popolnoma paralelno. Če bi algoritem implementirali na PRAM, ki predvideva neomejeno število procesorjev bi imel naš algoritem časovno zahtevnost $O(\log_2 n)$. Generator bi lahku preuredili tudi tako, da bi generiral za podatkovno-pretokovno arhitekturo. Zaradi nedavnih tehnoloških napredkov je podatkovno-pretokovna arhitektura postala konkurenčna CPUju in v mnogih primerih ga celo prehitela [8].

Poglavje 3

Štetje k-gramov

Za demonstracijo generativne algoritmike smo se odločili, da implementiramo štetje k-gramov. K-grami so strnjeno zaporedje k elementov v besedilu. Ti elementi so lahko posamezni znaki ali cele besede. Mi bomo šteli k-grame posameznih znakov. Taki k-grami so uporabni za prepoznavanje naravnih jezikov [6] in klasifikacijo genoma v družino genomov [13]. V bioinformatiki so k-grami znotraj genoma znani tudi kot k-meri. Njihovo štetje je zelo pomembno za to področje in zato je v ta namen razvitih že mnogo programov v ta namen. Primera takih programov sta Jellyfish [9] in BFCounter [11].

Eden izmed glavnih ciljev inženiringa algoritmov je narediti uporabno knjižnico z dobro strukturirano dokumentacijo. Zato smo si tudi mi zadali cilj, da ustvarimo hitro in uporabno knjižnico za štetje k-gramov. Knjižnica bo generirana v programska jezika C in Python. C je očitna odločitev zaradi njegove hitrosti, Python pa zaradi njegove fleksibilnosti in razširjenosti med podatkovnimi analitiki [1]. Python knjižnica bo dejansko samo ovoj okoli C knjižnice, ki ga bomo naredili s pomočjo Cython knjižnice.

3.1 Definicija problema

Definicija 1. Abeceda je končna neprazna množica simbolov in jo označujemo s Σ.

Med abecede tipično štejemo:

- binarna abeceda ($\Sigma = \{0, 1\}$),
- šestnajstiška abeceda ($|\Sigma| = 16$),
- DNK abeceda ($\Sigma = \{A, C, T, G\}$),
- angleška abeceda ($|\Sigma| = 26$).

Definicija 2. Niz s je končno urejeno zaporedje simbolov iz abecede Σ.

Dolžina niza s, označena kot $|s|$, je število vseh simbolov v nizu in velja $|s| \in \mathbb{N}_0$. Niz z dolžino 0 je prazen niz in ga označimo kot ε. Znak na i-tem mestu v nizu s označimo kot $s[i]$, kjer velja $i \in \{1, 2, \ldots, |s|\}$.

Množica vseh nizov nad abecedo Σ z dolžino n je Σ^n.

Definicija 3. $\Sigma^n = \{s_1, s_2, \ldots, s_{|\Sigma|^n}\}$, kjer $|s_i| = n \wedge s_i[j] \in \Sigma$, za $\forall i \in \{1, 2, \ldots, |\Sigma|^n\} \wedge \forall j \in \{1, 2, \ldots, n\}$

$\Sigma^0 = \{\varepsilon\}$ velja za vsako abecedo Σ. Na primer, če $\Sigma = \{A, B\}$, potem $\Sigma^1 = \{A, B\}$, $\Sigma^2 = \{AA, AB, BA, BB\}$,

$$\Sigma^3 = \{AAA, AAB, ABA, ABB, BAA, BAB, BBA, BBB\}$$

in tako naprej.

Podniz niza s je niz t sestavljen iz zaporednih znakov iz s in ohranjenim vrstnim redom.

Definicija 4. Podniz (ali faktor) niza $s = s[1] \ldots s[|s|]$ je niz $t = s[i] \ldots s[m+i]$, kjer velja $i \geq 1$ in $m + i \leq |s|$.

K-gram je podniz s fiksno dolžino k. Niz dolžine n ima lahko največ $n - k + 1$ različnih k-gramov.

Na primer, če vzamemo niz $s = banana$ in $k = 2$ dobimo naslednje podnize: ba, an, na, an in na. Nato je potrebno še prešteti število istih pojavitev in dobimo rezultat kot je prikazan v tabeli 3.1.

Podniz	Število pojavitev
ba	1
an	2
na	2

Tabela 3.1: Prešteti podnizi dolžine 2 niza *banana*

3.2 Splošni algoritem

Splošni algoritem smo implementirali z zgoščeno tabelo. Z njim bomo kasneje primerjali izvajalne čase generiranega algoritma. Algoritem 5 prikazuje psevdokodo algoritma. Parametra algoritma sta vhodni niz s in velikost gramov k. Rezultat algoritma je zgoščena tabela kgrams, ki vsebuje k-grame iz niza s in število njihovih pojavitev. Z infiksnim operatorjem to generiramo seznam vse možne indekse v mejah od 0 do $n - k + 1$, kjer je n dolžina niza s. V vsaki iteraciji vzame podniz na indeksih od i do $i + k$. Če je podniz že prisoten v zgoščeni tabeli, povečamo število njegovih pojavitev za 1, drugače ga dodamo v tabelo in inicializiramo število njegovih pojavitev na 1.

Algoritem 5 Štetje k-gramov s pomočjo zgoščene tabele

```
1: for i in (0 to n - k + 1) do
2:     kgram ← s[i:i+k]
3:     if kgram in kgrams then
4:         kgrams[kgram] ← kgrams[kgram] + 1
5:     else
6:         kgrams[kgram] ← 1
7:     end if
8: end for
```

Časovna zahtevnost Algoritma 5 je določena z enačbo 3.1. Časovna zahtevnost je odvisna od dveh parametrov n in k. V praksi za k velja $k \ll n$, torej lahko privzamemo $k = O(1)$. Zato ga lahko odstranimo iz prvega

faktorja, ki predstavlja število vseh iteracij algoritma. Drugi faktor pa predstavlja operacije nad zgoščeno tabelo, za katere privzamemo, da se izvedejo v konstantnem času [5, Poglavje 11].

$$T(n,k) = O(n-k+1) \cdot O(1) = O(n) \tag{3.1}$$

3.3 Generirani končni avtomat

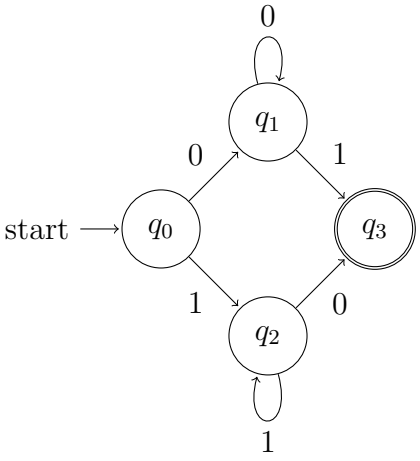
3.3.1 Končni avtomati

K-grame bomo šteli s pomočjo končnega avtomata stanj. Končni avtomat stanj je matematični model računanja, ki se uporablja za modeliranje računalniških programov in načrtovanje sekvenčnih logičnih vezij [7, Poglavje 1].

Avtomat je lahko natanko v enem stanju iz končne množice vseh stanj. Na začetku je v stanju, ki mu rečemo začetno stanje. Avtomat lahko prehaja med stanji, ko pride do nekega dogodka, ali je izpolnjen določen pogoj. Končni avtomat stanj je definiran s peterko $A = (Q, \Sigma, \delta, q_0, F)$, kjer so posamezne komponente definirane kot

1. Q je končna, neprazna množica stanj.
2. Σ je končna množica vhodne abecede.
3. δ je funkcija prehodov stanj $\delta : Q \times \Sigma \to Q$.
4. q_0 je začetno stanje avtomata in $q_0 \in Q$.
5. F je množica končnih stanj in $F \subseteq Q$.

Primer končnega avtomata stanj je prikazan na sliki 3.1.
Peterka komponent avtomata pa je sledeča:



Slika 3.1: Primer končnega avtomata stanj

- $Q = \{q_0, q_1, q_2, q_3\}$
- $\Sigma = \{0, 1\}$
- $q_0 = q_0$
- $F = \{q_3\}$

Funkcija prehodov stanj δ je prikazana s tabelo 3.2. Vsaka celica tabele predstavlja neko kombinacijo vseh simbolov vhodne abecede Σ in vseh stanj avtomata Q. Če je v celici zapisano /, pomeni, da avtomat ne more preiti v naslednje stanje za to specifično kombinacijo simbola in stanja.

δ	0	1
q_0	q_1	q_2
q_1	q_1	q_3
q_2	q_3	q_2
q_3	/	/

Tabela 3.2: Funkcija prehodnih stanj δ avtomata na sliki 3.1

3.3.2 Generirani avtomat

Za generiranje avtomata potrebujemo dva parametra: abecedo Σ in velikost gramov k. Iz njiju nato lahko izračunamo peterko komponent avtomata, kot je definirano v razdelku 3.3.1. Vhodna abeceda za generiranje avtomata Σ je enaka vhodni abecedi avtomata Σ. Začetno stanje $q_0 = \varepsilon$ in množica končnih stanj $F = \Sigma^k$. Množica vseh stanj je definirana z enačbo 3.2.

$$Q = \bigcup_{i=0}^{k} \Sigma^i = \{ \underbrace{q_0}_{\Sigma^0}, \underbrace{q_1, q_2, \ldots, q_{sum(1)-1}}_{\Sigma^1}, \ldots, \underbrace{q_{sum(k-1)}, \ldots, q_{sum(k)-1}}_{\Sigma^k} \} \quad (3.2)$$

Takšen zapis množice Q nam pomaga pri predstavitvi funkcije prehodov stanj δ. Za pomožno funkcijo smo definirali še funkcijo $\nu(x) = \sum_{i=0}^{x} |\Sigma|^i = \frac{|\Sigma|^{x+1}-1}{|\Sigma|-1}$. Funkcijo prehodov stanj δ zopet lahko predstavimo kot tabelo. Zato ločimo množico vseh stanj Q na dve podmnožici: stanja množice $Q_z = \bigcup_{i=0}^{k-1} \Sigma^i$ in stanja množice $Q_k = \Sigma^k$. Za indekse stanj množice Q_z velja

$$i \geq 0 \wedge i < \nu(k-1)$$

in za indekse stanj množice Q_k velja

$$i \geq \nu(k-1) \wedge i \leq \nu(k) - 1$$

, kjer $i \geq 0 \wedge i < |Q|$.

Takšna ločitev je potrebna, ker obe množici uporabljamo v drugačne namene in iz tega tudi sledijo drugačni prehodi stanj. Stanja množice Q_z poskrbijo, da avtomat pride v eno izmed stanj množice Q_k. Avtomat takrat lahko začne s štetjem k-gramov in se ne more več vrniti v stanja množice Q_z.

V tabeli 3.3 je prikazana tabela prehodov stanj za splošni avtomat. S horizontalno črto sta ločeni množici stanj Q_z in Q_k. Vrstica s stanjem q_i prikazuje izračun prehodov stanj za katerokoli izmed stanj množice Q_z. Prehode stanj množice Q_k pa sestavimo tako, da $|\Sigma|$-krat ponovimo prehode stanj množice Σ^{k-1}.

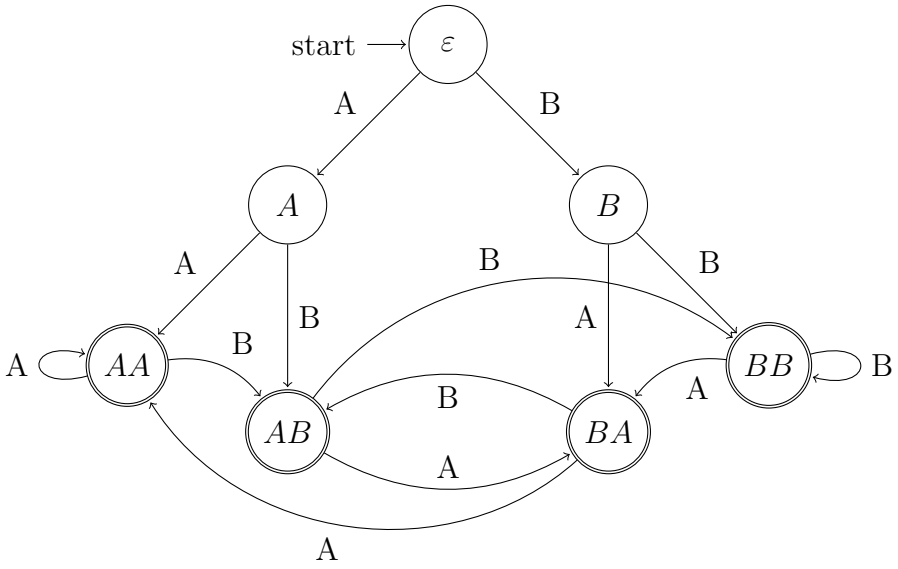
	δ	σ_1	$\cdots$	$\sigma_{\lvert\Sigma\rvert}$	
	q_0	q_1	$\cdots$	$q_{\lvert\Sigma\rvert}$	
	$\vdots$	$\vdots$	$\ddots$	$\vdots$	
	q_i	$q_{i\lvert\Sigma\rvert+1}$	$\cdots$	$q_{(i+1)\lvert\Sigma\rvert}$	
	$\vdots$	$\vdots$	$\ddots$	$\vdots$	
Σ^{k-1}	$q_{\nu(k-2)}$	$q_{\nu(k-2)\lvert\Sigma\rvert+1}$	$\cdots$	$q_{(\nu(k-2)+1)\lvert\Sigma\rvert}$	
Σ^{k-1}	$\vdots$	$\vdots$	$\ddots$	$\vdots$	
Σ^{k-1}	$q_{\nu(k-1)-1}$	$q_{(\nu(k-1)-1)\lvert\Sigma\rvert+1}$	$\cdots$	$q_{\nu(k-1)\lvert\Sigma\rvert}$	
Σ^{k-1}	$q_{\nu(k-1)}$	$q_{\nu(k-2)\lvert\Sigma\rvert+1}$	$\cdots$	$q_{(\nu(k-2)+1)\lvert\Sigma\rvert}$	$\lvert\Sigma\rvert$-krat
Σ^{k-1}	$\vdots$	$\vdots$	$\ddots$	$\vdots$	
Σ^{k-1}	$q_{\nu(k-1)+\lvert\Sigma\rvert}$	$q_{(\nu(k-1)-1)\lvert\Sigma\rvert+1}$	$\cdots$	$q_{\nu(k-1)\lvert\Sigma\rvert}$	
	$\vdots$	$\vdots$	$\ddots$	$\vdots$	
Σ^{k-1}	$q_{\nu(k)-1-\lvert\Sigma\rvert}$	$q_{\nu(k-2)\lvert\Sigma\rvert+1}$	$\cdots$	$q_{(\nu(k-2)+1)\lvert\Sigma\rvert}$	
Σ^{k-1}	$\vdots$	$\vdots$	$\ddots$	$\vdots$	
Σ^{k-1}	$q_{\nu(k)-1}$	$q_{(\nu(k-1)-1)\lvert\Sigma\rvert+1}$	$\cdots$	$q_{\nu(k-1)\lvert\Sigma\rvert}$	

Tabela 3.3: Tabelarična predstavitev funkcije prehodov stanj δ za splošni avtomat

V enačbi 3.3 je še matematični zapis funkcije δ. Funkcija kot prvi argument sprejme stanje $q_i \in Q$ in kot drugi argument sprejme simbol $\sigma_j \in \Sigma$. Rezultat funkcije je indeks naslednjega stanja $q_{\delta(q_i,\sigma_j)} \in Q$.

$$\delta(q_i, \sigma_j) = \begin{cases} |\Sigma|i + j + 1, & \textbf{če } i < \nu(k-1) \\ |\Sigma|((\nu(k-1) - |\Sigma|^{k-1}) + (i - \nu(k-1)) \mod |\Sigma|^{k-1}) + \\ j + 1, & \text{sicer} \end{cases} \tag{3.3}$$

Kot primer je na Sliki 3.2 prikazan končni avtomat za štetje 2-gramov niza, ki vsebuje samo simbola A in B.



Slika 3.2: Končni avtomat stanj

Avtomat lahko opišemo tudi z naslednjo peterko komponent:

- $\Sigma = \{A, B\}$
- $Q = \{\varepsilon, A, B, AA, AB, BA, BB\}$
- $q_0 = \varepsilon$
- $F = \{AA, AB, BA, BB\}$

V tabeli 3.4 so izračunani prehodi stanj.

		Σ	
Oznaka stanja	Stanje	A	B
ε	q_0	q_1	q_2
A	q_1	q_3	q_4
B	q_2	q_5	q_6
AA	q_3	q_3	q_4
AB	q_4	q_5	q_6
BA	q_5	q_3	q_4
BB	q_6	q_5	q_6

Tabela 3.4: Tabelarična predstavitev prehodov stanj končnega avtomata stanj

3.3.3 Generator avtomata

V tem poglavju bomo predstavili implementacijske podrobnosti generatorja avtomata, kot je prikazan v poglavju 3.3.2. Za predstavitev avtomata v programskem jeziku C smo izbrali programski konstrukt struct. Z njim smo definirali nov podatkovni tip z imenom kgrams_t. Njegovi atributi so vidni v izvorni kodi 3.1.

```
typedef struct _kgrams_t {
        unsigned int k;
        unsigned long alphabet_size;
        unsigned long state_count;
        int* transitions;
        int* char_map;
        char** state_names;
        int* result;
} kgrams_t;
```

Izvorna koda 3.1: Podatkovna struktura, ki opisuje končni avtomat stanj

Opis atributov tipa kgrams_t:

- k = velikost gramov k,
- alphabet_size = velikost abecede $|\Sigma|$,
- state_count = število stanj $|Q|$,
- transitions = funkcija prehodov stanj δ,
- char_map = preslikava iz simbola v število, ki predstavlja njegov indeks v abecedi Σ,
- state_names = imena stanj Q,
- result = tabela preštetih k-gramov velikosti $|Q|$.

Definicija je kgrams_t v zaglavni datoteki kgrams.h. Zaglavna datoteka je datoteka s končnico .h, ki vsebuje C funkcije, makro definicije in konstante, ki jih uporabnik lahko vključi v lasten program.

V algoritmu 6 so pomožne funkcije, ki jih bomo uporabili v glavnem algoritmu. V funkcijah uporabljamo funkcijo fill(x, y), ki vrne seznam velikosti x in vsebuje y. Funkcija getStateNames generira oznake vseh stanj. Seznam oznak inicializiramo s seznamom, ki vsebuje samo prazni niz ε in tako lahko nastavimo začetno mejo zanke na 1 namesto 0. Vsaka iteracija zanke doda Σ^i na konec seznama state_names. Σ^i pomeni generiranje permutacij s ponavljanjem dolžine i abecede Σ. Oznake stanj niso ključne za izvajanje avtomata, ampak so pomembne pri vračanju rezultatov. Oznaka stanja na nekem indeksu i bo imela zapisano število njenih pojavitev na istem indeksu v tabeli result. Naslednja funkcija je getTransitions in vrne seznam prehodov stanj. Funkcija generira množici Q_z in Q_k, in ju vrne združene v en seznam. Zadnja funkcija getCharMapping nam bo pomagala pri preslikavi iz znakovne predstavitve simbola v njegov indeks znotraj abecede Σ. Pri preslikavi bomo uporabili ASCII tabelo. Tabelo char_map bomo inicializirali z velikostjo 128, enako kot ASCII tabela. Vse vrednosti na začetku nastavimo na število -1, s katerim bomo zaznali znake, ki niso del vhodne abecede Σ. Sedaj moramo vse simbole iz abecede Σ pretvoriti v njihovo ASCII predstavitev, ki je

število od 0 do 127. To število nam predstavlja index v naši tabeli char_map, na katerega zapišemo indeks simbola. Na primer $\Sigma = \{A, B\}$:

$$char\_map[65] = 0,$$

$$char\_map[66] = 1.$$

Algoritem 6 Pomožne funkcije za glavni algoritem

```
1: function getStateNames(k, Σ)
2:     state_names ← [ε]
3:     for i in (1 to k+1) do
4:         state_names += Σ^i
5:     end for
6:     return state_names
7: end function
8: function getTransitions(|Q|, |Σ|)
9:     Q_z ← 1 to |Q|
10:    last_level = []
11:    for i in (1 to |Σ|^k) do
12:        last_level += (i + ν(k − 1))
13:    end for
14:    Q_k ← fill(|Σ|, last_level)
15:    return Q_z + Q_k
16: end function
17: function getCharMapping(Σ)
18:    char_map ← fill(128, -1)
19:    for i in (0 to Σ) do
20:        char_map[Σ[i].toInt] ← i
21:    end for
22:    return char_map
23: end function
```

Algoritem 7 uporabi definirane pomožne funkcije, da generira avtomat.

Vsebuje glavno funkcijo generateKgramsAutomaton, ki kot argumenta sprejme velikost k-gramov in vhodno abecedo Σ. Končni rezultat funkcije je zaglavna datoteka, ki vsebuje inicializirano spremenljivko tipa kgrams_t. Iz obeh vhodnih argumentov izračunamo število vseh stanj $|Q|$ v vrstici 2. V naslednjih treh vrsticah kličemo definirane pomožne funkcije in inicializiramo:

- seznam prehodov stanj δ v vrstici 3,
- preslikavo simbolov v število char_map v vrstici 4,
- oznake stanj state_names v vrstici 5.

Preostane nam še inicializacija tabele result, ki bo hranila preštete pojavitve k-gramov. Na tem mestu imamo inicializirane vse spremenljivke in jih lahko zapišemo v zaglavno datoteko.

Algoritem 7 Generiranje avtomata za štetje k-gramov

1: **function** GENERATEKGRAMSAUTOMATON(k, Σ)
2: $|Q| \leftarrow \sum_{i=0}^{k} |\Sigma|^i$
3: $\delta \leftarrow$ GETTRANSITIONS($|Q|, |\Sigma|$)
4: char_map $\leftarrow$ GETCHARMAPPING(Σ)
5: state_names $\leftarrow$ GETSTATENAMES(k, Σ)
6: result $\leftarrow$ fill($|Q|, 0$)
7: WRITETOFILE(k, $|\Sigma|$, $|Q|$, δ, char_map, state_names, result)
8: **end function**

Glavna funkcija algoritma je funkcija generateKgramsAutomaton, ki kot argumenta sprejme velikost k-gramov in abecedo Σ. V vrstici 2 izračunamo število vseh stanj v avtomatu in skupaj z velikostjo abecede Σ podamo funkciji za izračun prehodov stanj getTransitions.

Ostane nam še inicializacija tabele result z velikostjo $|Q|$ v vrstici 6. Na koncu zapišemo vse potrebne atribute kgrams_t strukture v zaglavno datoteko v vrstici 7.

V izvorni kodi 3.2 je prikazan primer zaglavne datoteke z avtomatom, kot je prikazan na sliki 3.2.

```
kgrams_t GENOME_K_GRAM = {
2,
7,
(int[14]){1,2,3,4,5,6,3,4,5,6,3,4,5,6},
(int[128]){-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
-1,0,1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1},
(char*[7]){"","A","B","AA","AB","BA","BB"},
(int[7]){0,0,0,0,0,0,0}
};
```

Izvorna koda 3.2: Primer generirane zaglavne datoteke z avtomatom

3.3.4 Analiza zahtevnosti

Sledi analiza zahtevnosti algoritma 7. Kot je vidno iz glave funkcije v vrstici 1, bo časovna zahtevnost algoritma odvisna od velikosti k in vhodne abecede Σ. Algoritem ni zapleten, saj je sestavljen iz preprostih operacij izvedenih v linearnem času. Najbolj zahtevna operacija je v vrstici 4 algoritma 6, kjer generiramo oznake vseh stanj. V enačbi 3.4 je časovna analiza generatorja.

$$T(\Sigma, k) = \sum_{i=1}^{k} |\Sigma|^i = \frac{|\Sigma|(|\Sigma|^k - 1)}{|\Sigma| - 1} = O(|\Sigma|^{k+1}) \tag{3.4}$$

Ker so vsi ostali deli algoritma linearni, lahko privzamemo, da je časovna

zahtevnost algoritma $O(|\Sigma|^{k+1})$.

3.4 Algoritem za izvajanje avtomata

Algoritem 8 zna izvesti končni avtomat A, da prešteje k-grame v nizu s.

Algoritem 8 Štetje k-gramov s pomočjo avtomata stanj

```
1: function countKgrams(A, s)
2:     state ← 0
3:     for ch in s do
4:         ch_idx ← A.char_map[ch.toInt]
5:         if ch_idx < 0 then Error          ▷ Invalid input character
6:         state ← A.transitions[state · A.alphabet_size + ch_idx]
7:         A.result[state] += 1
8:     end for
9: end function
```

Algoritem vedno inicializira začetno stanje avtomata v q_0, ki predstavlja prazen niz ε. Algoritem iterira čez vsak znak vhodnega niza s in ga hrani v spremenljivki char. V vsaki iteraciji je potrebno pretvoriti char v njegovo številsko predstavitev s pomočjo avtomatove char_map tabele. Shranimo jo v spremenljivko next. Če je next večji od 0, pomeni, da $char \in \Sigma$. V nasprotnem primeru je char neveljaven znak in algoritem zaključi z izvajanjem. Tabelo transitions, ki predstavlja funkcijo prehodov stanj δ smo zasnovali kot enodimenzionalno, v poglavju 3.3.2 pa kot dvodimenzionalno tabelo. Zato moramo sami poskrbeti, da pretvorimo indeksa dvodimenzionalne tabele state in next v indeks enodimenzionalne tabele. Ker je prva dimenzija dvodimenzionalne tabele velika, $|\Sigma|$ z njo pomnožimo state in produktu prištejemo next. Vsota predstavlja indeks enodimenzionalne tabele, ki ga uporabimo, da iz tabele transitions dobimo naslednje stanje in ga shranimo v state. Dobljenemu stanju povečamo število pojavitev za 1 v tabeli result.

Algoritem je v datoteki libkgrams.c, prototip funkcije pa je zapisan v kgrams.h zaglavni datoteki.

3.5 Izboljšava algoritma

3.5.1 Generirani avtomat

Za izboljšavo algoritma je bila ključna ugotovitev, da množico stanj $Q' = \bigcup_{i=0}^{k-1} \Sigma^i$ potrebujemo samo za prvih k znakov vhodnega niza. Po k znakih končamo v enem izmed končnih stanj množice F, kjer avtomat tudi ostane do konca vhodnega niza. Zato namesto da generiramo množico stanj Q', sami poiščemo stanje v katerem bo avtomat po k znakih vhodnega niza.

Lastnosti izboljšanega avtomata so naslednje:

- $Q = \Sigma^k$,
- $F = \Sigma^k$.

Začetno stanje q_0 bo algoritem za izvajanje avtomata moral vsakič znova izračunati iz začetnih k znakov vhodnega niza. Funkcija prehodov stanj δ pa se močno poenostavi, kot je prikazano v tabeli 3.5.

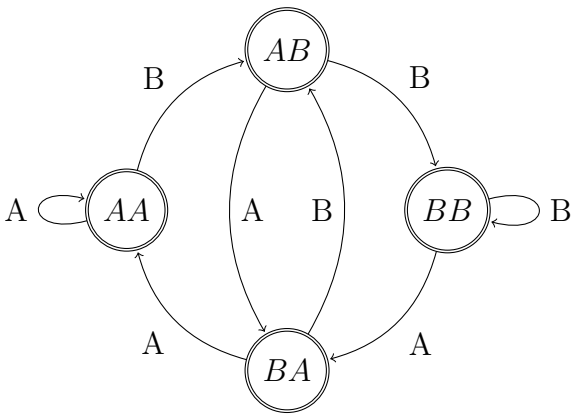
δ	Σ				
q_0	q_0	$\cdots$	$\cdots$	$q_{\lvert\Sigma\rvert-1}$	
$\vdots$	$\vdots$	$\ddots$	$\ddots$	$\vdots$	
$q_{\lvert\Sigma\rvert-1}$	$q_{\lvert Q\rvert-\lvert\Sigma\rvert-1}$	$\cdots$	$\cdots$	$q_{\lvert Q\rvert-1}$	
$\vdots$	$\vdots$	$\ddots$	$\ddots$	$\vdots$	$\lvert\Sigma\rvert$-krat
$q_{\lvert Q\rvert-\lvert\Sigma\rvert}$	q_0	$\cdots$	$\cdots$	$q_{\lvert\Sigma\rvert-1}$	
$\vdots$	$\vdots$	$\ddots$	$\ddots$	$\vdots$	
$q_{\lvert Q\rvert-1}$	$q_{\lvert Q\rvert-\lvert\Sigma\rvert-1}$	$\cdots$	$\cdots$	$q_{\lvert Q\rvert-1}$	

Tabela 3.5: Tabela funkcije prehodov stanj δ

Tabelo prehodov stanj dobimo tako, da vzamemo vrstice stanj od q_0 do $q_{|\Sigma|-1}$ in jih ponovimo še $|\Sigma| - 1$-krat. V teh vrsticah je zapisano zaporedje stanj z indeksi od 0 do $|Q| - 1$.

Kot primer lahko pretvorimo avtomat na sliki 3.2 v izboljšanega na sliki 3.3. Lastnosti avtomata:

- $\Sigma = \{A, B\}$
- $Q = \{AA, AB, BA, BB\}$,
- $F = \{AA, AB, BA, BB\}$.



Slika 3.3: Izboljšani generirani avtomat

V tabeli 3.6 je prikazana tabelarična predstavitev funkcije prehodov stanj δ.

3.5.2 Generiranje avtomata

Algoritem za generiranje izboljšanega avtomata je predstavljen v algoritmu 9. Funkcija getCharMapping deluje enako kot pri osnovnem algoritmu 7, zato je njena implementacija izpuščena. Ostali dve pomožni funkciji getTransitions in getStateNames se poenostavita v vrstici 3 in 5.

		Prehodi med stanji	
Oznaka stanja	Številka stanja	A	B
AA	0	0	1
AB	1	2	3
BA	2	0	1
BB	3	2	3

Tabela 3.6: Tabelarična predstavitev končnega avtomata stanj

Algoritem 9 Generiranje avtomata za štetje k-gramov

```
1: function generateKgramsAutomaton(k, Σ)
2:     |Q| ← |Σ|^k
3:     δ ← fill(|Σ|, 0 to |Q|)
4:     char_map ← getCharMapping(Σ)
5:     state_names ← Σ^k
6:     result ← fill(|Q|, 0)
7:     writeToFile(k, |Σ|, |Q|, δ, char_map, state_names, result)
8: end function
```

3.5.3 Izvajanje avtomata

V algoritmu 10 je prikazan algoritem za izvajanje izboljšanega avtomata. Funkcija countKgrams ima isto glavno zanko kot algoritem 8 za štetje k-gramov. Razlikujeta se v inicializaciji začetnega stanja. Pri osnovnem algoritmu je začetno stanje vedno 0, ki ponazarja prazen niz ε. Pri izboljšanem algoritmu pa kličemo funkcijo findInitialState, ki najde ustrezno začetno stanje. Iterira čez vse oznake stanj avtomata A in jih primerja s prvimi k znaki vhodnega niza s. Ko na nekem indeksu i najde ujemanje, ta indeks vrne v funkcijo countKgrams in ga uporabi za začetno stanje.

Algoritem 10 Izboljšano štetje k-gramov s pomočjo avtomata stanj

```
1: function countKgrams(A, s)
2:     start ← s[0:A.k]
3:     state ← findInitialState(A, start)
4:     for ch in s do
5:         ch_idx ← A.char_map[ch.toInt]
6:         if ch_idx < 0 then Error                ▷ Invalid input character
7:         state ← A.transitions[state · A.alphabet_size + ch_idx]
8:         A.result[state] += 1
9:     end for
10: end function
11: function findInitialState(A, s)
12:     for i in (0 to A.state_count) do
13:         if A.state_names[i] == s then
14:             return i
15:         end if
16:     end for
17: end function
```

3.6 Uporaba generirane knjižnice

Ker je celoten postopek generiranja in prevajanja avtomata zapleten, smo ga hoteli poenostaviti z uporabo orodja GNU make. Make je avtomatizacijsko orodje, ki samodejno zgradi izvršljive programe in knjižnice iz izvorne kode. To naredi tako, da prebere datoteko imenovano Makefile, ki vsebuje navodila, kako dobimo ciljni program.

V našem primeru Makefile za svoje delovanje potrebuje naslednje parametre:

- velikost k-gramov,
- vhodno abecedo Σ,
- ime zaglavne datoteke,
- ime spremenljivke tipa kgrams_t,
- tip algoritma.

Tip algoritma lahko zavzame vrednosti 0, 1 in 2. Vrednost 0 uporabimo za osnovno različico avtomata, vrednost 1 za generiranje izboljšane različice, vrednost 2 pa za generiranje Python modula, ki je opisan v naslednjem razdelku.

Kot primer lahko zopet vzamemo avtomat s Slike 3.2 in z njim izdelamo program za štetje bigramov znakov A in B. V izvorni kodi 3.3 je prikazana uporaba make ukaza, ki generira tak avtomat.

```
make generator \
        K=2 \
        ALPHABET=AB \
        OUTPUT_FILE=countbigrams.h \
        AUTOMATON_NAME=KGRAMS_AUTOMATON \
        ALG_VERSION=0
```

Izvorna koda 3.3: Generiranje knjižnice

Make ukaz izstavi zaglavno datoteko z imenom countbigrams.h, ki vsebuje spremenljivko tipa kgrams_t z imenom KGRAMS_AUTOMATON. V izvorni kodi 3.5 je demonstrirana uporaba generirane knjižnice. V header datoteki kgrams.h je definirana metoda count_kgrams, ki sprejme kot argumenta generirani avtomat in niz. Program na koncu izpiše vse k-grame, ki so se pojavili vsaj enkrat.

```
#include <stdio.h>
#include "kgrams.h"
#include "countbigrams.h"

int main(int argc, char const *argv[]) {
  char* input = "ACTGGTCA";
  count_kgrams(KGRAMS_AUTOMATON, input);
  // Print the non-zero counts
  for (int i = 0; KGRAMS_AUTOMATON.state_count; i++) {
    if (KGRAMS_AUTOMATON.result[i] != 0) {
      printf("bigram: \%s, Count: \%d\n",
        KGRAMS_AUTOMATON.state_names[i],
        KGRAMS_AUTOMATON.result[i]);
    }
  }
  return 0;
}
```

Izvorna koda 3.4: Uporaba generirane knjižnice

3.6.1 Python knjižnica

Za generiranje Python knjižnice lahko uporabimo isti ukaz, kot je naveden v izvorni kodi 3.3, le da spremenimo $ALG\_VERSION = 2$.

```
from count2grams import GENOME_K_GRAM
counts = GENOME_K_GRAM.count_kgrams(b"ACTGGTCA")
for bigram, count in counts.items():
        if count != 0:
                print(bigram, count)
```

Izvorna koda 3.5: Uporaba generirane knjižnice

Poglavje 4

Rezultati

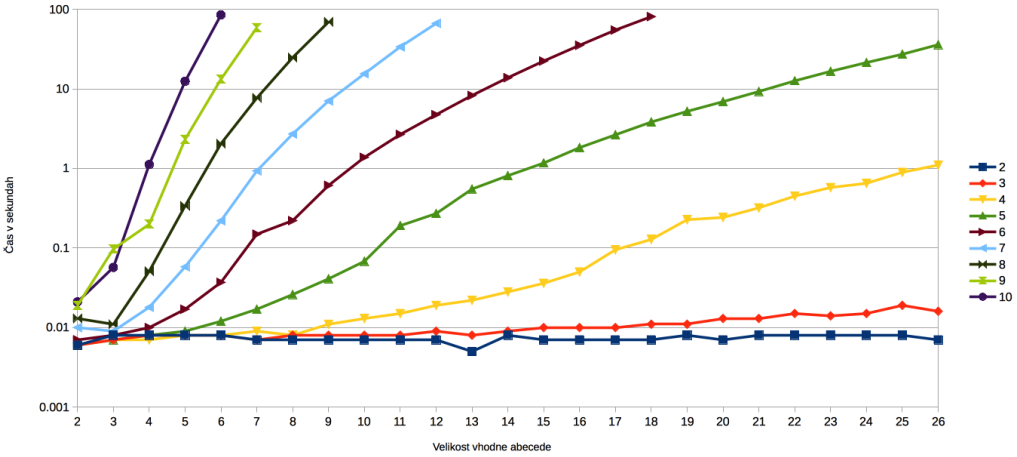
Za eksperimentalno evaluacijo smo primerjali izvajalne čase, čase prevajanja programov in velikost generiranih datotek vseh štirih algoritmov. Testiranje je potekalo na računalniku Macbook Pro z naslednjimi specifikacijami:

- operacijski sistem: OSX El Capitan 10.11.5,
- procesor: 2,7 GHz Intel Core i5,
- predpomnilnik: 3MB deljeni L3,
- pomnilnik: 8 GB 1867 MHz DDR3.

4.1 Izvajalni časi generatorjev avtomatov

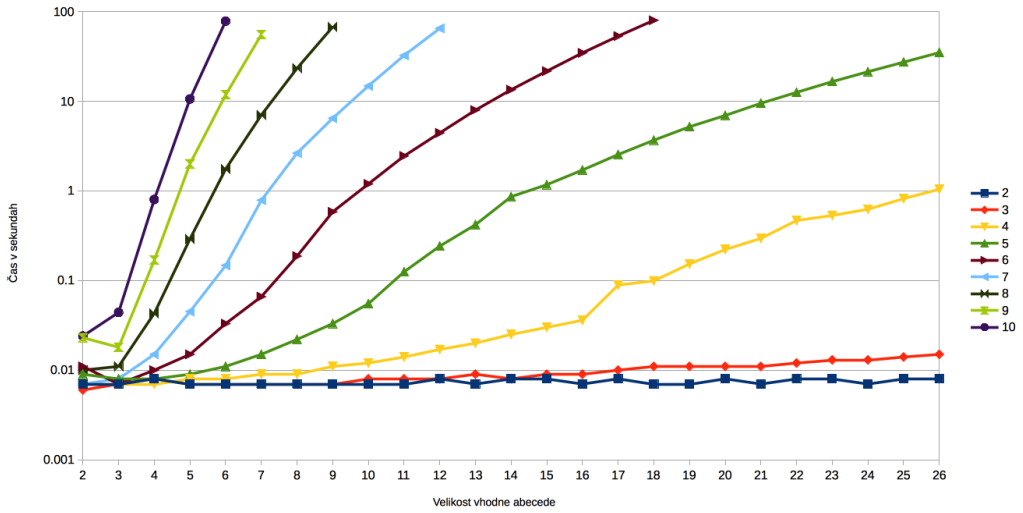
Merili smo izvajalne čase generatorja osnovnega in izboljšanega avtomata. Čase generatorja Python modula nismo merili, ker je v osnovi isti kot generator izboljšanega avtomata s prilagojenim izpisom za programski jezik Python. Za testni množici smo si izbrali velikost k-gramov $k = \{2, 3, \ldots, 10\}$ in velikosti abeced $|\Sigma| = \{2, 3, \ldots, 26\}$. Množico velikosti k-gramov nismo hoteli izbrati preveliko, ker smo že vnaprej predvidevali, da bi nastajali problemi pri dolgih časih generiranja in pri velikosti zaglavnih datotek. V množici velikosti abeced pa smo hoteli vključiti dve uporabni velikosti: 4 za bazne

pare v genomu in 26 za angleško abecedo. Omejili smo tudi čas generiranja na 100 sekund, da ne bi nastajale prevelike zaglavne datoteke, ki bi jih bilo kasneje težko prevesti. Na sliki 4.1 so prikazani izvajalni časi generatorja osnovnega avtomata v odvisnosti od števila simbolov v abecedi. Različne barve črt pomenijo različno velik k, za katerega smo pognali eksperiment. V legendi so navedene velikosti k in njihove barve v grafu. Problemi za generator se začnejo, ko k postane večji od 5. Do takrat generator uspe generirati vse avtomate v roku 100 sekund.



Slika 4.1: Čas generiranja osnovnega algoritma (y os je logaritemska)

Rezultati istega poskusa za izboljšani avtomat so na sliki 4.2. Izboljšani algoritem ni dosegel pričakovanih časovnih rezultatov. Ima podobne čase generiranja kot osnovni algoritem, le pri večjih k-jih se pozna manjša časovna razlika v prid izboljšanega algoritma.

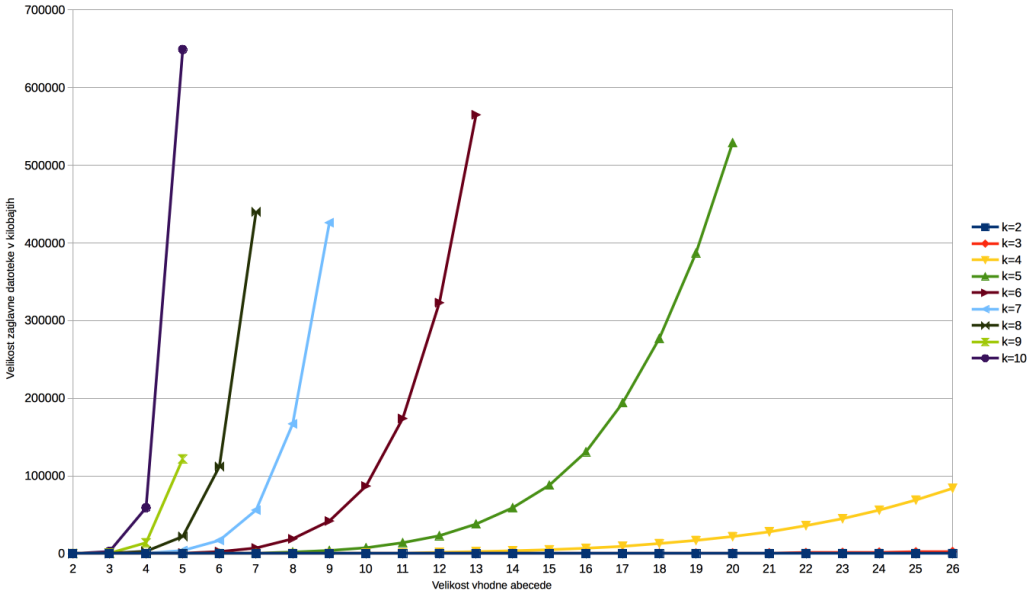


Slika 4.2: Čas generiranja izboljšanega algoritma (y os je logaritemska)

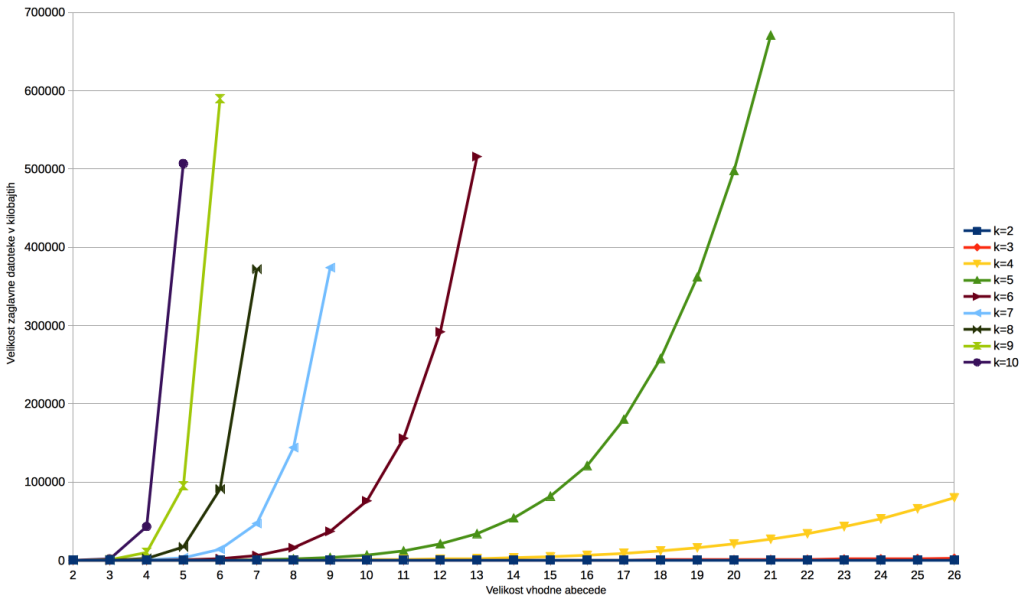
4.2 Velikost generiranih zaglavnih datotek

Naprej nas je zanimala velikost generiranih zaglavnih datotek, ki sta jih generirala oba algoritma iz prejšnjega razdelka. Maksimalno velikost datoteke smo omejili na 700 MB. Na sliki 4.3 so prikazane velikosti zaglavnih datotek, ki jih je generiral osnovni algoritem. Prikazane so velikosti zaglavnih datotek v odvisnosti od števila simbolov v abecedi. Pri majhnih velikostih abeced in k-gramov so velikosti datotek obvladljive. Takoj, ko povečamo obe dimenzije, velikosti rastejo eksponentno. Če bi radi šteli k-grame različnih velikosti za veliko abecedo Σ, bi za to porabili veliko prostora na disku.

Isti eksperiment smo ponovili še za izboljšani avtomat. Pri majhnih velikostih k zaglavne datoteke niso bistveno manjše v primerjavi z osnovnim avtomatom. Ko pa večamo k, so razlike tudi po več kot 100 MB.



Slika 4.3: Velikosti zaglavnih datotek osnovnega avtomata



Slika 4.4: Velikosti zaglavnih datotek izboljšanega avtomata

4.3 Časi prevajanja končnih programov

Za naslednji eksperiment smo merili čas prevajanja minimalnega C programa z vključenimi generiranimi zaglavnimi datotekami. Upoštevane zaglavne datoteke so bile tiste, ki so bile generirane v manj kot 100 sekundah, in njihova velikost je manjša od 700MB. Primer minimalnega C programa je v izvorni kodi 4.1 in shranili smo ga v datoteko z imenom main.c.

```
int main(){return 0;}
```

Izvorna koda 4.1: Minimalni C program

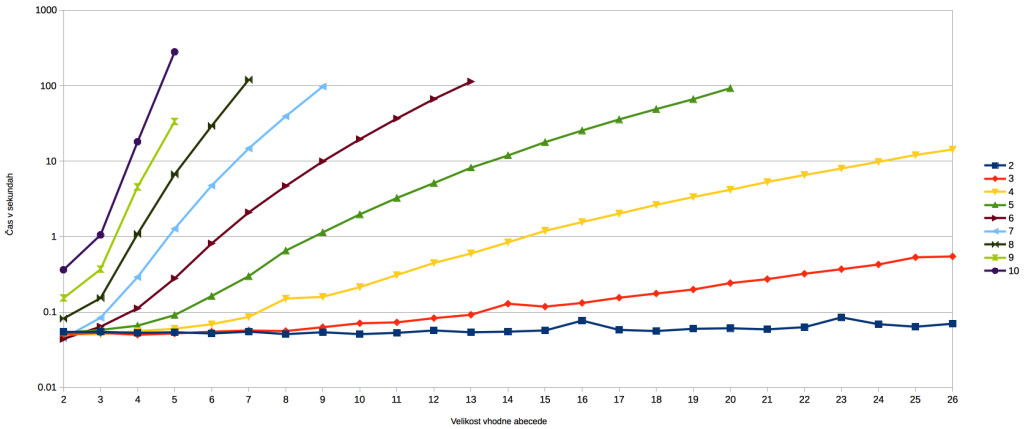
Prevajali smo s pomočjo prevajalnika gcc kot je prikazano v izvorni kodi 4.2. Zaglavno datoteko kgrams.h, ki vključuje definicijo avtomata, in zaglavno datoteko, ki vsebuje sam avtomat smo vključili preko argumenta -include v ukazni vrstici.

```
gcc -O3 -Wall -o main main.c -include"kgrams.h"
                             -include"pot/do/avtomata"
```

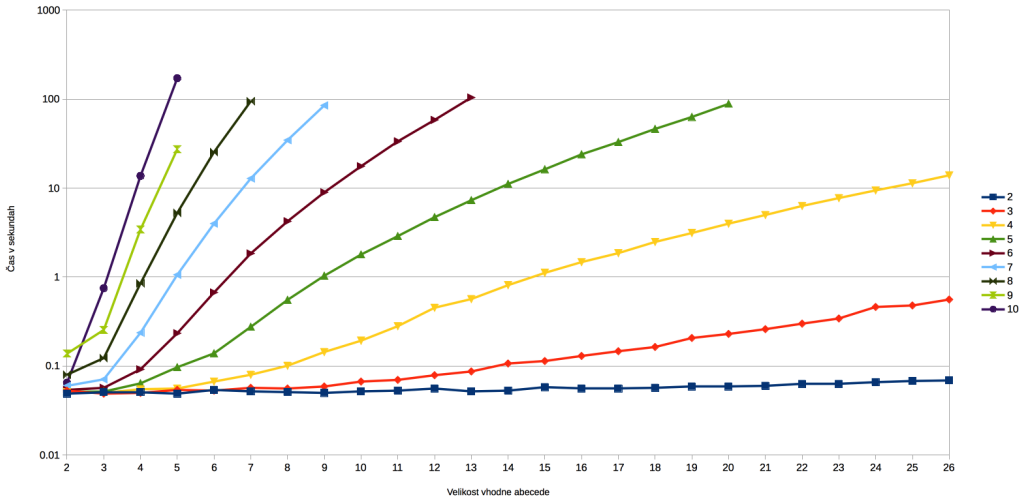
Izvorna koda 4.2: Prevajanje minimalnega C programa z vključenim avtomatom

Časi prevajanja za osnovni avtomat so vidni na sliki 4.5. Ker velikosti zaglavnih datotek naraščajo eksponentno, naraščajo podobno tudi časi prevajanja.

Časi prevajanja za izboljšani avtomat so vidni na sliki 4.6. Podobno kot pri velikosti zaglavnih datotek iz prejšnjega razdelka se prednosti izboljšanega avtomata vidijo šele pri višjih k-jih.



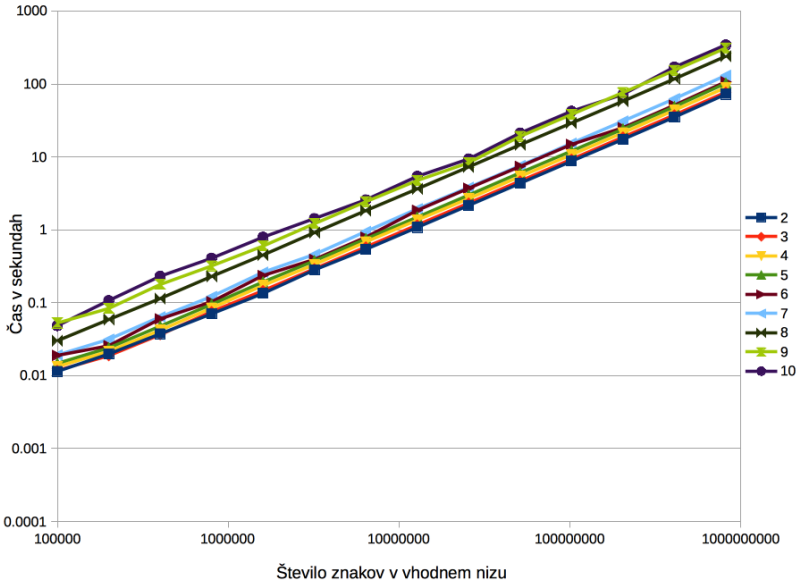
Slika 4.5: Časi prevajanja zaglavnih datotek osnovnega avtomata



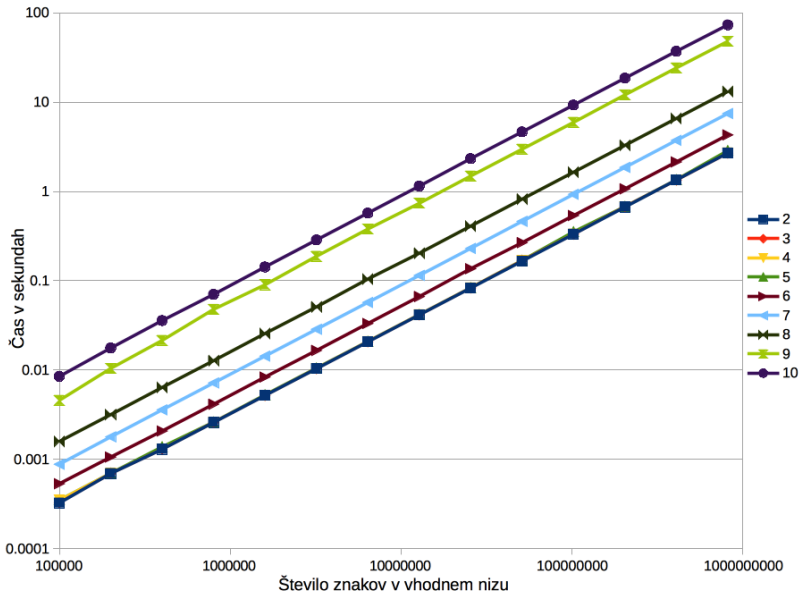
Slika 4.6: Časi prevajanja zaglavnih datotek izboljšanega avtomata

4.4 Izvajalni časi končnih programov

Glavni eksperiment so izvajalni časi programov, ki uporabijo generirane avtomate, da preštejejo k-grame v določenem nizu. Za testiranje smo vzeli abecedo velikosti $|\Sigma| = 4$, ki predstavlja bazne pare v genomu. Izbrali smo jo zato, ker v njej vidimo še največji potencial za uporabo našega algoritma. Za začetek bomo prikazali izvajalne čase splošnega algoritma in algoritma, ki uporablja naš osnovni generirani avtomat. Rezultati pri $|\Sigma| = 4$ za splošni algoritem so vidni na sliki 4.7, za algoritem z osnovnim avtomatom pa na sliki 4.8. Vidimo lahko, da sta oba algoritma linearne narave, s tem da časi algoritma z osnovnim avtomatom naraščajo bistveno počasneje.



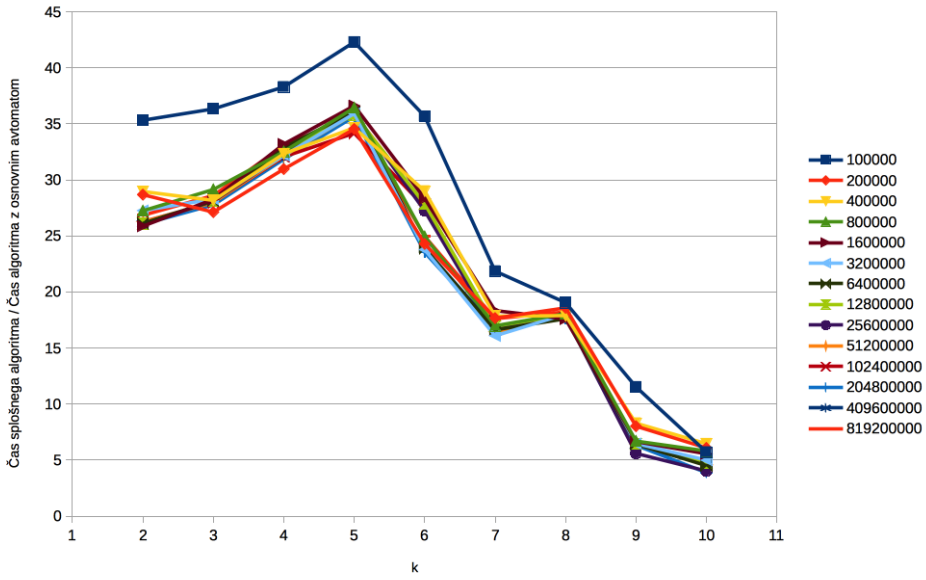
Slika 4.7: Izvajalni časi za splošni algoritem pri $|\Sigma| = 4$ (log-log graf)



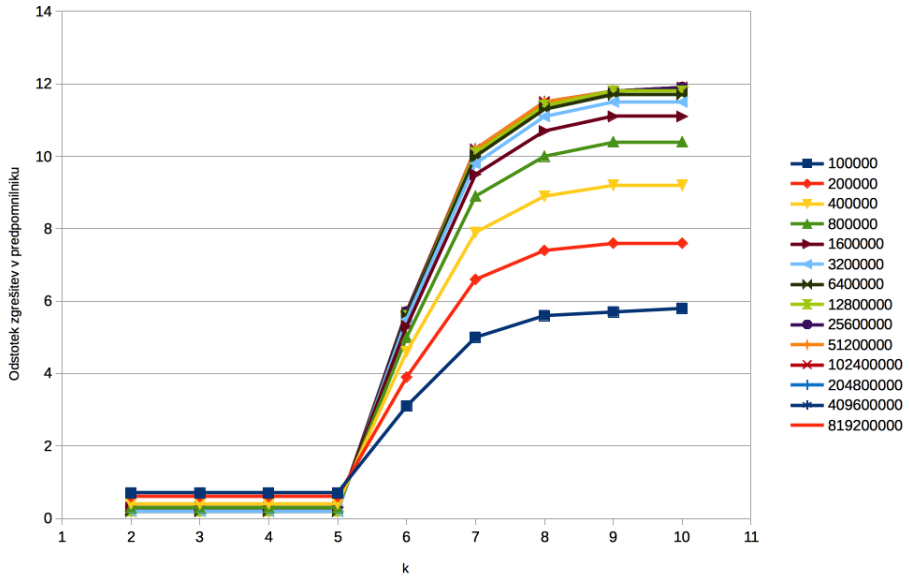
Slika 4.8: Izvajalni časi za algoritem z osnovnim avtomatom pri $|\Sigma| = 4$ (log-log graf)

Razliko med obema algoritmoma lahko najbolje pokažemo z grafom pohitritve. Na sliki 4.9 so vidni rezultati. Na x-osi je prikazana vrednost k, na y-osi pa razmerje med izvajalnim časom splošnega algoritma in izvajalnim časov algoritma z osnovnim avtomatom. Različne barve črt pomenijo različno velikost vhoda in njihove oznake so navedene v legendi grafa. Pohitritev pri vseh vhodih obetavno narašča do $k = 5$, kjer algoritem v povprečju doseže 36-kratno pohitritev. Potem pa pohitritev začne strmo padati in konča v povprečju pri 5-kratni pohitritvi pri $k = 10$. Do strmega padanja pohitritve najverjetneje pride zaradi vse večje velikosti avtomatov in zaradi tega pride do večjega števila zgrešitev v predpomnilniku.

To smo preverili z orodjem cachegrind in izmerili odstotek zgrešitev v predpomnilniku. Rezultati so vidni na sliki 4.10. V skladu z našimi napovedmi, vidimo, da v točki $k = 5$ pride do porasta v zgrešitvah v predpomnilniku. To potrdi naša predvidevanja, da pride do zmanjšanja pohitritve zaradi zgrešitev v predpomnilniku.

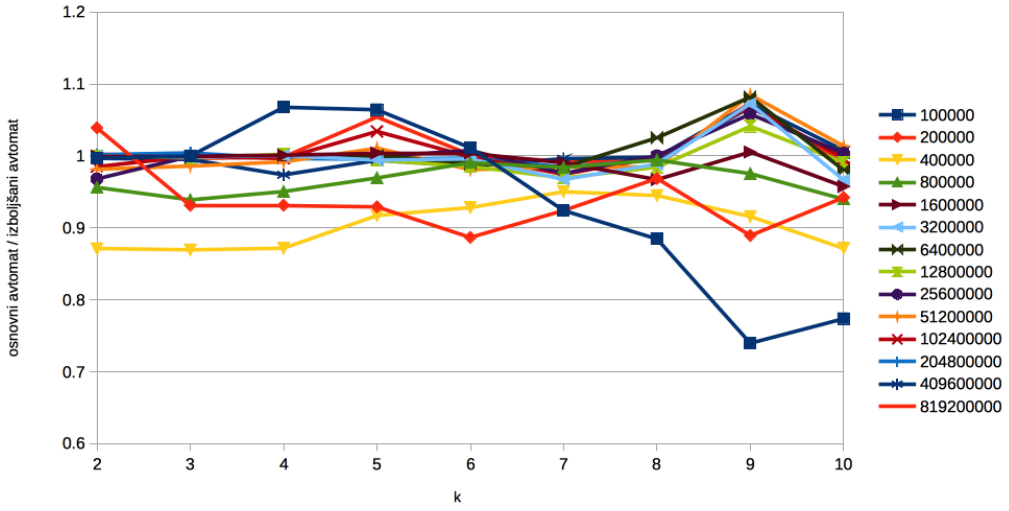


Slika 4.9: Pohitritev algoritma z osnovnim avtomatom v primerjavi s splošnim algoritmom

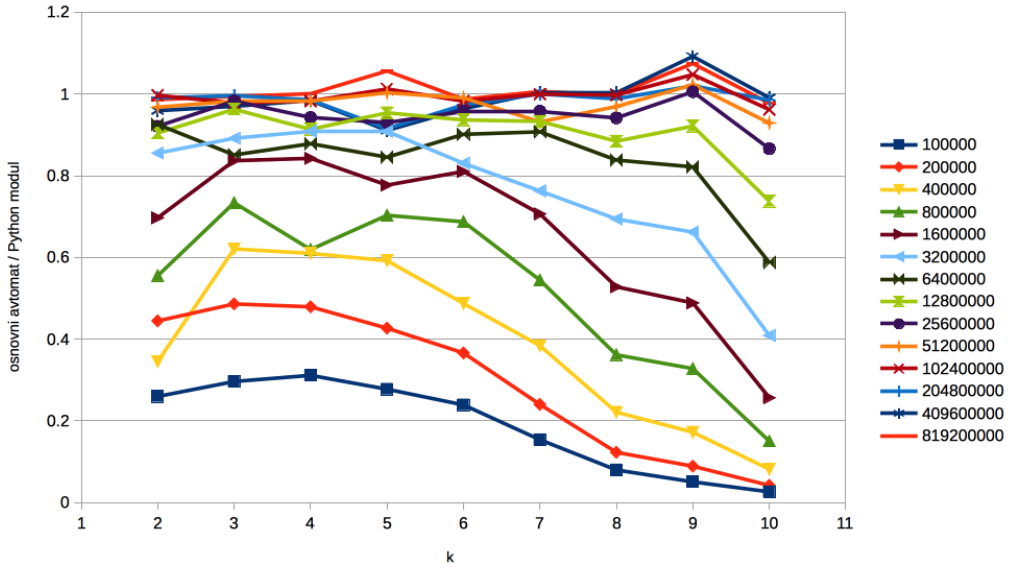


Slika 4.10: Odstotek zgrešitev v predpomnilniku pri izvajanju algoritma z osnovnim avtomatom

Za konec nas je zanimalo izvajanje algoritma z izboljšanim avtomatom in Python modula. Oba smo primerjali z algoritmom, ki uporablja osnovni avtomat. Hoteli smo videti, ali smo kaj izgubili s tem, ko smo zmanjšali število stanj in avtomat napisali kot Python modul. Na sliki 4.11 so rezultati primerjave algoritma z osnovnim avtomatom in algoritma z izboljšanim avtomatom. Odstopanja od razmerja 1 so minimalna, kar pomeni, da s tem, ko smo zmanjšali število stanj nismo izgubili na času, prihranili pa smo pri prostoru za hranjenje avtomata. Na sliki 4.12 pa je vidna primerjava algoritma z osnovnim avtomatom in Python modulom. Tukaj vidimo, da je Python modul pri manjših velikostih vhodnega niza časovno izrazito slabši. Razlog je v tem, da mora Python modul, preden lahko začne izvajati Cjevski program pretvoriti lastne podatkovne tipe v tiste, ki jih uporablja programski jezik C. Python modul se časovno izenači šele pri večjih velikostih vhodnega niza, ko čas porabljen za izvajanje samega algoritma pretehta čas za pretvorbo tipov.



Slika 4.11: Razmerje med algoritmom z osnovnim avtomatom in algoritmom z izboljšanim avtomatom



Slika 4.12: Razmerje med algoritmom z osnovnim avtomatom in Python modulom

Poglavje 5

Zaključek

V diplomski nalogi smo predstavili generativno algoritmiko, področje znotraj inženiringa algoritmov in prikazali njeno uporabo. Osnovne pristope smo prikazali pri reševanju paralelnega računanja predpn. Za glavni problem smo si izbrali štetje k-gramov. Na začetku definiramo terminologijo k-gramov in implementiramo splošni algoritem za štetje k-gramov, s katerim bomo kasneje primerjali generirane rešitve. Nato definiramo še končne avtomate stanj na katerih so osnovane naše generirane rešitve. Prikazali smo teoretično zasnovo in implementacijo generatorjev tako osnovnega kot izboljšanega avtomata. Za obe različici avtomatov smo morali tudi implementirati različne algoritme, ki jih izvajajo. Končamo z demonstracijo generiranja in uporabe generirane knjižnice za štetje k-gramov. Sledi analiza časov generiranja avtomatov, velikosti generiranih zaglavnih datotek, čas njihovega prevajanja in najbolj pomembno izvajalni čas algoritmov, ki uporabijo generirane avtomate.

Analiza štetja k-gramov je prikazala dobre in slabe strani generativne algoritmike. Problemi so izhajali iz eksponentne narave algoritma, kar je pomenilo dolge čase generiranje, velike končne datoteke in dolge čase prevajanja. Ampak to so vse enkratni stroški, ker prevedeno knjižnico lahko uporabimo velikokrat pod pogojem, da je ta bistveno hitrejša od splošne rešitve. Za našo rešitev štetja k-gramov se je izkazalo, da je hitrejša od splošne rešitve,

ampak samo pri majhni vrednosti k. To izhaja iz velikih tabel prehodov stanj avtomata, ki so bile prevelike za predpomnilnik. To bi lahko ublažili z uporabo manjših podatkovnih tipov, kjer bi bilo možno. Tako bi samo zamaknili problem za nekaj k, ne bi ga pa dokončno rešili. V neki točki bi se še zmeraj izkazalo, da se nam bolj splača sprotno izračunavanje prehodov stanj avtomata, namesto da uporabljamo že izračunane.

Zanimivo bi bilo preizkusiti generativno algoritmiko še na več algoritmih in preizkusiti več pristopov generiranja. Naslednje zanimivo področje so deli-in-vladaj algoritmi, pri katerih bi lahko sploščili nekaj zadnjih nivojev rekurzije. Tukaj bi zopet morali iskati točko preloma med hitrostjo algoritma in velikostjo generirane programske kode.

Literatura

[1] R, python duel as top analytics, data science software – kdnuggets 2016 software poll results. `http://www.kdnuggets.com/2016/06/r-python-top-analytics-data-mining-data-science-software.html`. Dostopano: 2016-07-19.

[2] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 2001.

[3] Guy E Blelloch. Prefix sums and their applications. 1990.

[4] Markus Chimani and Karsten Klein. *Experimental Methods for the Analysis of Optimization Algorithms*, chapter Algorithm Engineering: Concepts and Practice, pages 131–158. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[6] Ted Dunning. Statistical identification of language, 1994.

[7] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[8] Anton Kos, Sašo Tomažič, Jakob Salom, Nemanja Trifunovic, Mateo Valero, and Veljko Milutinovic. New benchmarking methodology and

programming model for big data processing. *Int. J. Distrib. Sen. Netw.*, 2015:4:4–4:4, January 2015.

[9] Guillaume Marçais and Carl Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, 2011.

[10] Catherine C. McGeoch. *A guide to experimental algorithmics*. Cambridge University Press, 32 Avenue of the Americas, New York, 2012.

[11] Páll Melsted and Jonathan K. Pritchard. Efficient counting of k-mers in dna sequences using a bloom filter. *BMC Bioinformatics*, 12(1):1–7, 2011.

[12] Peter Sanders. *Efficient Algorithms: Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*, chapter Algorithm Engineering – An Attempt at a Definition, pages 321–340. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[13] Andrija Tomović, Predrag Janičić, and Vlado Kešelj. n-gram-based classification and unsupervised hierarchical clustering of genome sequences. *Computer Methods and Programs in Biomedicine*, 81(2):137–153, 2016/07/16.